JPL Publication 88-25

A Communication Channel Model of the Software Process

Robert C. Tausworthe

October 15, 1988

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No.
JPL Publication 88-25

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
A Communication Channel Model of the Software Process

5. Report Date
October 15, 1988

6. Performing Organization Code
JPL

7. Author(s)
R.C. Tausworthe

8. Performing Organization Report No.
JPL Publication 88-25

9. Performing Organization Name and Address

JET PROPULSION LABORATORY
California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109

10. Work Unit No.

11. Contract or Grant No.
NAS7-918

12. Sponsoring Agency Name and Address

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Washington, D.C. 20546

13. Type of Report and Period Covered
JPL Publication

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This publication reports beginning research into a noisy communication channel analogy of software development process productivity, in order to establish quantifiable behavior and theoretical bounds. The analogy leads to a fundamental mathematical relationship between human productivity and the amount of information supplied by the developers, the capacity of the human channel for processing and transmitting information, the software product yield (object size), the work effort, requirements efficiency, tool and process efficiency, and programming environment advantage. The publication also derives an upper bound to productivity that shows that software reuse is the only means that can lead to unbounded productivity growth; practical considerations of size and cost of reusable components may reduce this to a finite bound.

17. Key Words (Selected by Author(s))

Communications
Computer Programming and Software
Information Theory
Operations Research

18. Distribution Statement

Unclassified/Unlimited

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of Pages

22. Price

JPL Publication 88-25

A Communication Channel Model of the Software Process

Robert C. Tausworthe

October 15, 1988

NASA

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade, name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

Abstract

This publication reports beginning research into a noisy communication channel analogy of software development process productivity, in order to establish quantifiable behavior and theoretical bounds. The analogy leads to a fundamental mathematical relationship between human productivity and the amount of information supplied by the developers, the capacity of the human channel for processing and transmitting information, the software product yield (object size), the work effort, requirements efficiency, tool and process efficiency, and programming environment advantage. The publication also derives an upper bound to productivity that shows that software reuse is the only means that can lead to unbounded productivity growth; practical considerations of size and cost of reusable components may reduce this to a finite bound.

Contents

List of Figures

1 Introduction

As Boehm [1] notes in a recent article, the computer software industry for years has been accused of inferior productivity in comparison to its hardware counterpart, whose productivity continues to increase at an intense rate. Despite advances in languages, development environments, workstations, methodologies, and tools, software projects seem to continue to grind out production-engineered code at about the same old 8 to 15 delivered lines of source code per staff-day. Yet, as Boehm also points out, if software is judged using the same criteria as hardware, its productivity looks pretty good. One can produce a million copies of a developed software product as inexpensively as a million copies of a computer hardware product. The area in which productivity has been slow to increase is the development and sustaining phases of the software life cycle.

Profit-making organizations may amortize their software development and sustaining costs over large customer markets, so that low development productivity is mitigated by larger and larger markets. But government agencies, their contractors, and non-profit organizations must rely on increases in productivity to avoid costs and improve quality. Development and sustaining costs are not often recovered by duplicating the product many, many times.

Software development and sustaining productivity has been the subject of many articles to date. It is also the focus of this publication, which is, in a sense, a mathematical proof of Brooks' [2] assertion that "there is no silver bullet." The avenues for productivity improvement have been adequately summarized by Boehm [1] as

1. Get the best from people.
2. Make the process more efficient.
3. Eliminate steps where possible.
4. Stop reinventing the wheel.
5. Build simpler products.
6. Reuse components.

All of Boehm's steps above, except the first, are human-information-input reductive. Software tools, aids, support environments, workstations, office automation, automated documentation, automated programming, front-end aids, knowledge-based assistants, information hiding, modern programming practices, life-cycle models, common libraries, application generators, next-generation languages, *etc.* all save labor by supplying or modifying information at a faster rate or more reliably than can be done by humans.

Software is information for computers that is made from information supplied by people. Some of the human input information may be new, and some may be reused, perhaps altered for the new application. Some of the output information product is thus new, and some may derive from legacy, perhaps altered for the function intended. It is therefore intuitive to think of productivity in terms of the amount of information appearing in the output product relative to the effort required from humans to supply the needed information relating to that product. We shall more precisely define productivity using this concept a little later; for the present, let us merely acknowledge that software production capacity increases when the effort required from humans in supplying the information needed to construct a given product is reduced.

It is reasonable, then, to put information and communication theory to work on the theoretical capacity of productivity. In 1949, Claude Shannon [3] proved that communications channels have theoretical information transmission rate limits that are influenced by their channel configurations, signal-to-noise ratios, and bandwidths[1]. Humans and computers developing software are communications devices and channels, and therefore subject to Shannon's law. *Humans are capable of transmitting information only at a rate below their capacity limit* [4]. The channel may transmit more data volume than the actual number of *information bits* due to redundancy and encoding; however, the information rate of bits emanating from the output (*i.e.*, the output entropy) may not exceed the rate that information bits are input (the input entropy). In the parlance of information theory and thermodynamics, there can be no "Maxwell's demons" in the channel.

When building an information product, part of the input information needed is in the form of "black box" specifications of functional and performance requirements. Some of this is new, supplied by humans, and some of it is old, retrieved from other existing sources. But which portions of the old information are to be reused, and how they are to be located, extracted, modified, and integrated with the new information comprises more new information that also must be (largely) supplied by humans.

Once a new or modified software product has been developed, both it and its components are candidates for reuse in forthcoming software products. Thus, the repertoire of reusable objects may grow without bound as the industry wends its way into the future. Reusable objects may be envisioned as new functions appended to an extensible implementation language that may be used in the next project. The conceptual minimum information required at the human input interface is merely that required to select the language features to be used and to integrate them properly into the operating product(s). In the ideal, we may look to automated and knowledge-based tools to supply the other necessary searching, manipulative, transformational, and inferential information associated with matching function-to-language-feature correspondences, integration and construction of the product, and validation.

The question arises, then, can the information content of the output products

[1] The most popular form [3] of Shannon's law is $C_0 = B\log_2(1 + S/N)$.

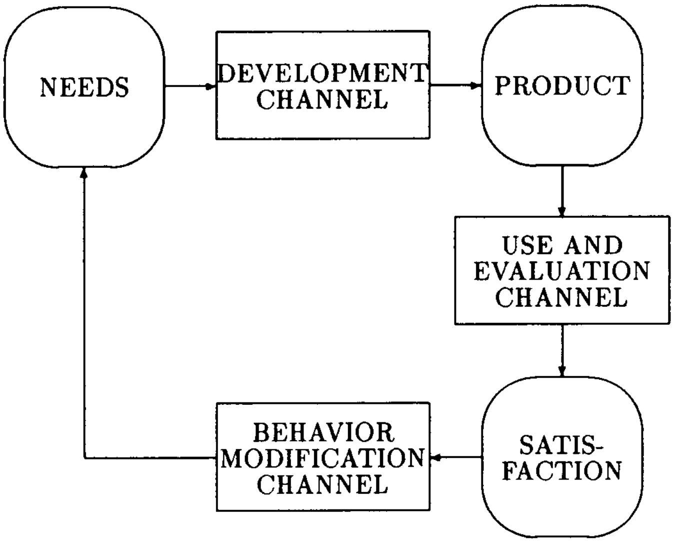

Figure 1: An abstract product life cycle process.

in such an ideal software environment continue to grow at a faster rate than the input rate, or is productivity growth limited by some form of "Shannon limit?" If so, what are the factors which control that limit? This publication develops a framework for answering these questions and characterizing the solutions.

2 The Communication Analogy

The discussion above characterizes the software development process as one in which, as in Figure 1, various kinds of information are supplied by humans toward implementing a product whose form is also information: documents, programs, parametric data, databases, and test data. Software development is thus an Information-Input/Information-Output (I^3O) process. In like fashion, the use and evaluation of software products are also I^3O processes. Even the behavior modification that shapes needs based on the level of satisfaction derived from use and evaluation of the products is, to some extent, an I^3O process.

An I^3O process may thus be portrayed, for purposes here, as a noisy communication channel with the following traits:

1. *Transformational.* Output information (*i.e.*, the product) exists in a different form than provided in the input (*i.e.*, requirements).
2. *Distortive.* Some input requirements may be implemented differently than intended.

3. *Erasive.* Some of the input requirements may not have been implemented.

4. *Spurious.* Some features implemented may not have been specified in the input requirements.

5. *Random delay.* The transport time from requirements to product is a variable time, only partially predictable.

6. *Random cost to use.* The cost in dollars and effort needed to transform requirements into products is only partly predictable. The cost of products is the cost of operating the channel.

7. *Non-stationary.* The uncertainty aspects of the channel vary with time.

As is true of other communications systems, the channels themselves must be constructed before they can be used, at a certain cost. I^3O channels consist of people and machines working in randomly connected orchestration. Moreover, the I^3O channels that are used to construct products are themselves the products of other I^3O channels. Thus, if carried too far, the analogy becomes more intricately interconnected, complex, and difficult to analyze, but perhaps more true to life.

Software problems restated in terms of I^3O channels are:

- channel costs are too high.
- throughput delay is too long.
- input/output correlation is too low and difficult to validate.
- input and output are not entirely quantifiable, consistent, nor tangible.
- cost, delay, and throughput are not entirely predictable nor controllable.

More microscopically, an overall communications channel may be viewed as an interconnected network of noisy components and sub-channels. In analogy, high-level software problems decompose into smaller interrelated contributory problems, deriving from many sources. During the conceptualization, requirements capture, and alignment processes of the product cycle, distortion and noise (faults) appear as the result of unknown or unrecognized needs, unexpressed needs, wrongly expressed needs, conflicting needs, non-stationary needs, and inability to quantify and articulate needs. During the implementation and alteration stages, noise comes from misunderstood or ambiguous requirements, conflicting views of utility, inability to simulate a product in entirety, inadvertent omission, conflicting requirements, and unfeasible requirements. During the testing and validation stage, difficulties arise in the combinatorial impracticality of certainty, in the need for an operational environment in some actual or simulated form, in the need for the product in a simulated or completed, mature form, and in the need for definitive acceptance criteria. Ultimately, the evaluation and enlightenment processes require products and operational environments in completed or simulated form, and are exposed to imprecise, subjective, intangible satisfaction criteria.

Typical considerations which relate to, contribute, or cause these problems are the complexity of the I^3O channels and the products they produce, the stochastic behavior of people, and rapidly changing hardware and software technology. Moreover, our understanding of the software process is still in its evolutionary stage: tools, environments, and systems are only moderately sophisticated. Methods, models, and theoretical bases for development and product analyses are sparse and largely invalidated. Preparation of products for legacy has often not been properly consummated during development. The reuse of inheritance has been difficult, even when legacy goals are adequately set and fulfilled. Automated knowledge bases for software engineering and applications domains are in their infancy. The transmission medium (*i.e.*, human language) lacks precision in many contexts. And, finally, the skill base of software personnel has not yet been adequately oriented to a disciplined, standardized, industrial-strength engineering approach.

Feedback is commonly used in electronics to stabilize performance. However, the high costs and long delays in I^3O channel usage tend to inhibit firm, immediate feedback for risk of fomenting an unstable situation and incurring yet higher implementation costs and longer delays.

The communication system approach to improvement of channel performance, however, is simple and straightforward:

1. Measure and characterize the channel and its parameters.
2. Expect transmission to be distorted, noisy, and delayed, and provide appropriate compensation.
3. Design the information throughput rate to be within channel capacity (as, e.g., Shannon's limit, or other formula applying to the particular channel[2]).
4. Remove redundancy in the source information before transmission.
5. Make the transmitted information be resilient to channel disturbances by using effective encoding and decoding techniques.
6. Transmit information through the channel with as great a signal force as possible.
7. Take steps to reduce disturbances within the communications channel.
8. Use feedback to correct errors.

[2] Software production capacity in the absence of fault generation and correction is given by Eq. 25.

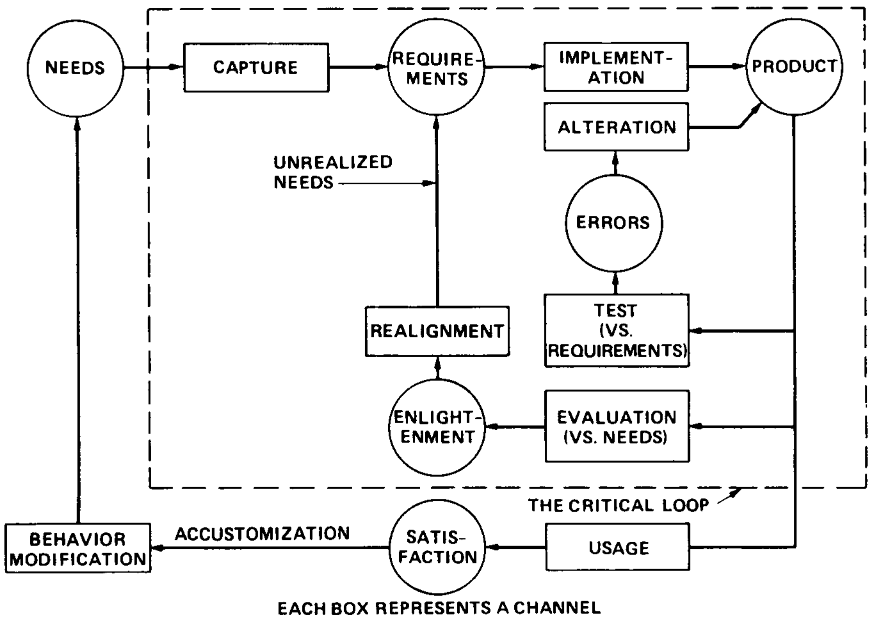

Figure 2: The I^{3}O life-cycle channel model.

The goal of this publication, then, is to characterize and quantify software production in analogy with communications theory, and thereby in terms of measurable, causal, and controllable factors.

3 The Software Channels

A basic idealized production configuration was depicted in Figure 1, where needs are faithfully projected in the form of information through the development channel to yield information products, which are then used, evaluated, and may lead to a certain level of satisfaction. Use and accustomization beget behavior modification, which, in turn, elevates the original set of needs toward higher levels of automation. Not present in this ideal are the intrinsic distortions, faults, and other flaws that produce less-than-ideal products, incomplete levels of satisfaction, and, perhaps, unfortunate modifications of behavior that limit the tendency toward higher automation.

A refinement of this concept is shown in Figure 2, where the processes associated with channel imperfections are displayed more prominently. Needs are projected through a capture channel to produce a requirements specification;

requirements are transmitted via an implementation channel into the product set; the product set is put through a testing channel to reveal (some of the) errors; errors are fed into the alteration channel, which (partially) corrects the product set; evaluation of the product set against stated requirements often reveals shortcomings, leading to an enlightened state; and enlightenment guides the process of requirements realignment. Usage of the product set, as earlier, produces a level of satisfaction (not necessarily complete), which alters the state of need through behavior modification.

Each of the information transmission channels and information sets can be further dissected and detailed for better understanding of the transformation processes and better accuracy in modeling the software phenomena.

The critical, and perhaps less philosophical, portion of the refined software channel analogy is shown inside the dashed lines of Figure 2. This portion comprises the software development and sustaining segments of the life cycle. Note that the analogy can be made to simulate information transmission aspects of the "ordinary waterfall" life cycle, incremental development, rapid prototyping, evolutionary enhancement, and "spiral" life cycle paradigms merely by suitable definitions of channel characteristics. In the next section, the software channel analogy is used to develop a refinery model of software productivity, to which information and communication theory are applied to derive statistical limitations on human capacity to produce larger and larger software systems.

4 The Implementation Channel

The assumed software implementation components are illustrated in Figure 3. Five forms of information input by humans are identified: requirements (function, performance, and constraints), transformational (design and coding), combinational (integration), corroborative (validation and verification), and management (status and control). Each of these potentially contains imperfections in the form of accidents (inadvertent, random faults) and distortions (deliberate, non-random faults). Together, these latter two constitute a sixth type of information input by humans that we shall collectively refer to as *noise*. Also shown is the set of products resulting from the inputs.

Generation and application of the above input information to the software implementation channel is assumed to constitute the entire expenditure of human effort. Information generated by humans is mental, verbal, and documentation, and only the last of these is amenable to measurement. We must, therefore, hypothesize that the capture of information in memoranda, documents, code and comments, parametric and test data, *etc.*, is representative of and correlates significantly with the total outlay of effort.

Output products are viewed as condensations, transformations, and refinements of the information that came into the environment; hence, we refer to the implementation process as the **Software Refinery**. Productivity improvement

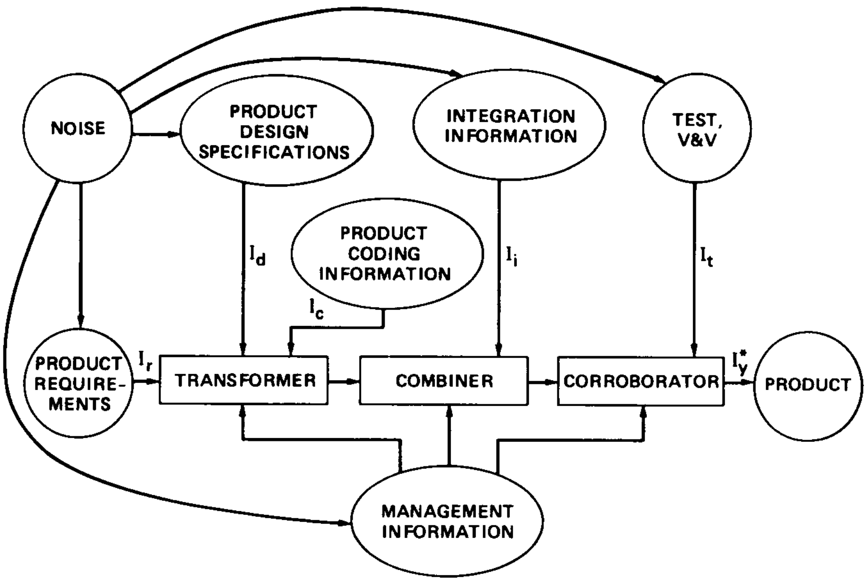

Figure 3: The software production refinery.

in the refinery is tantamount to reducing the amount of human-supplied input information required for a given output product set.

Effort-intensive input information requirements will be minimized by eliminating redundancy and by reusing existing information whenever feasible. For example, if a system has a requirement for a word processor of a known type, then the single expression "Wordstar[3] 4.0" could be used to convey unambiguously all the characteristics that the cited word processor possesses. Moreover, if there were only $1024 = 2^{10}$ word processors in the world, only 10 bits would be needed to distinguish Wordstar among its competitors. Only exceptional and incremental information would be then be needed to specify a slightly different capability desired. Additionally, since Wordstar already exists, further information relating to design, implementation, and testing is not required, except where it relates to the integration of that package into the system being built.

Also, when documents must be developed to contain previously generated information (*i.e.*, "boilerplate"), the only information conceptually required from the human is where to find the boilerplate, how much of it to use, where to put it, and any necessary alterations.

For the remainder of this publication, we shall focus on that information

[3] Wordstar is a registered trademark of MicroPro, Inc.

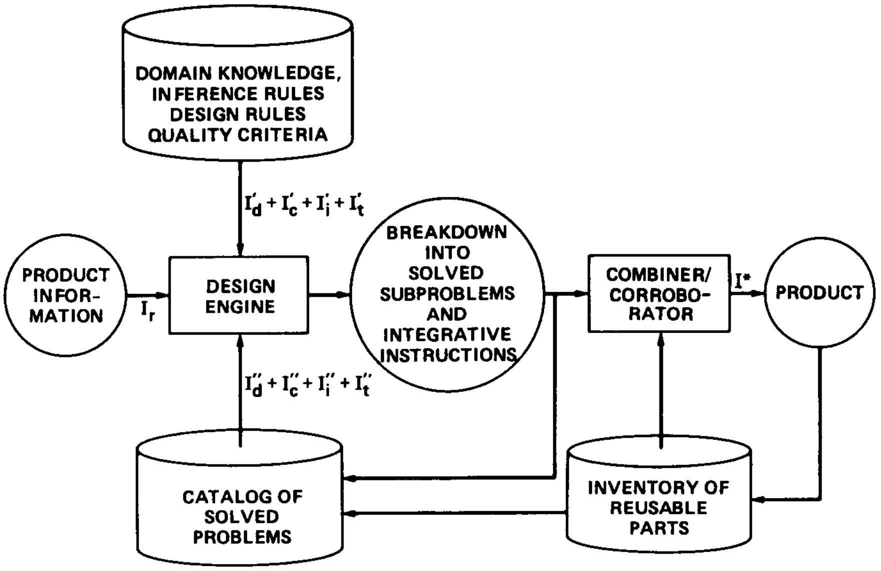
DOMAIN KNOWLEDGE,
INFERENCE RULES
DESIGN RULES
QUALITY CRITERIA
$I'_d + I'_c + I'_i + I'_t$
PRODUCT INFOR-MATION
I_r
DESIGN ENGINE
BREAKDOWN INTO SOLVED SUBPROBLEMS AND INTEGRATIVE INSTRUCTIONS
COMBINER/ CORROBO-RATOR
I^*
PRODUCT
$I''_d + I''_c + I''_i + I''_t$
CATALOG OF SOLVED PROBLEMS
INVENTORY OF REUSABLE PARTS

Figure 4: The ideal software refinery configuration.

leading to the program (set), or *product yield*. Therefore, effort and information used to produce documents is limited to that which is yield related. These include requirements documents, design specifications, project plans and status reports, test plans and procedures, and the like; preplanning, applications, operations, and maintenance documents are excluded at this time. We have hypothesized that the information content of these entities correlates strongly with the total project information. By measuring the information contents of software project documents and output yields, then, quantitative relationships among input information and output yield may be established.

Transformational and corroborative information input needs are potentially reduced by reusing previous designs and code whenever feasible. In the ideal, fully automated case, this reduction could be almost complete: automated catalogs of solved problems would be searched using knowledge bases having extensive application domain-dependent inference and design rules that match functional and performance requirements with known solutions and designs, designs with working code, *etc.* In the ideal automated software refinery, the amount of input noise, and thus the need for corroborative information, could also be drastically reduced. The ideal software refinery is shown in Figure 4.

Although much of the integrative information would also conceptually be supplied by automation, some will nevertheless still be required from humans to relate interdependency among functional features, data flows, and orders of precedence.

We model the software production refinery in the form of an extensible language. That is, the human information input $\mathcal{I}$ is used to develop the output yield $\mathcal{Y}$ from new information and from instructions to reuse existing information and previously developed parts that operate within given time and data precedence constraints.

The distinguished components of the input $\mathcal{I}$ are (Figure 3)

$$\mathcal{I} = \mathcal{I}_r \cup \mathcal{I}_d \cup \mathcal{I}_c \cup \mathcal{I}_i \cup \mathcal{I}_t \cup \mathcal{I}_m \tag{1}$$

These terms represent, respectively, requirements, design, code production, integration, test (including validation and verification), and management information sets. Each of the input sets potentially contains faulty information, or noise.

In particular, we shall assume that the requirements term, $\mathcal{I}_r$, can be isolated to contain the functional, performance, and algorithmic specifications and constraints, so that, in concept, a fully automated programming environment could produce the output yield in the current refinery without further information.

We define the ***inherent product specification***, $\mathcal{I}^*$, as the least practical information required to specify the output yield uniquely. It is the mapping of the input information through the production transformation

$$\pi(\mathcal{I}) = \mathcal{I}^* \tag{2}$$

Conversely, that subset of the input, denoted $\hat{\mathcal{I}}$, that traces to the as-built product is defined by the inverse production transform,

$$\pi^{-1}(\mathcal{I}^*) = \hat{\mathcal{I}} \tag{3}$$

Note that this traceability may not necessarily be direct: constraints, performance requirements, and design goals in $\mathcal{I}$ certainly influence the resulting $\mathcal{I}^*$; but it may be difficult indeed to correspond any tokens of the output product with tokens of the input information. Therefore, $\hat{\mathcal{I}}$ should be regarded as that (amended) form of $\mathcal{I}$ that got built.

The sets of fulfilled and unfulfilled requirements are described by

$$\mathcal{I}_f = \mathcal{I}_r \cap \hat{\mathcal{I}} \tag{4}$$

and

$$\mathcal{I}_e = \mathcal{I}_r - \mathcal{I}_f \tag{5}$$

respectively. That is, $\mathcal{I}_f$ is that portion of $\mathcal{I}_r$ that got implemented, and $\mathcal{I}_e$ is the remainder of $\mathcal{I}_r$.

The executable program, or ***apparent yield*** $\mathcal{Y}$ will include the inherent product specification, $\mathcal{I}^*$, as well as the $\mathcal{I}_i^*$ of each of the n modules in the refinery invoked by $\mathcal{I}^*$, as transformed by the compiler and linker into a functioning unit. $\mathcal{Y}$ will normally be sensitive to compiler and linker characteristics, such as type and degree of code optimization, extent of program and data segmentation, *etc.* Thus, we define the ***inherent functional yield***, $\mathcal{Y}^*$, as the join of inherent product specifications over all components comprising the final product,

$$\mathcal{Y}^* = \bigcup_{i=0}^{n} \mathcal{I}_i^* \tag{6}$$

in which $\mathcal{I}_0^* = \mathcal{I}^*$.

We denote the sizes of these sets by

$$\begin{aligned}
I_k &= |\mathcal{I}_k| \quad \text{for } k = r,d,c,i,t,m & (7)\\
I &= |\mathcal{I}| \quad \leq I_r + I_d + I_c + I_i + I_t + I_m & (8)\\
I^* &= I_0^* = |\mathcal{I}^*| & (9)\\
I_i^* &= |\mathcal{I}_i^*| \quad \text{for } i = 1,\ldots,n & (10)\\
Y &= |\mathcal{Y}| & (11)\\
Y^* &= \sum_{i=0}^{n} I_i^* & (12)
\end{aligned}$$

Naturally, $I^* \leq I_r$ by Shannon's law, and *a fortiori* $I^* \leq I$. Also, $I^* \leq Y^*$ because $\mathcal{I}^* \subseteq \mathcal{Y}^*$.

Input information is perhaps most meaningfully measured in terms of the *chunks* [4] that humans treat as units of information in memory and recall. However, the mechanism for chunking is not yet well enough understood (at least, by the author) to be able to compute an input information chunk measure. Rather, the first-order entropy [3] based on word and symbol, or *token*, counts and vocabulary usage will be used:

$$\begin{aligned}
H_k &= -\sum_{i=1}^{R_k} p_{k,i} \log_2 p_{k,i} \qquad \text{for } k = r,d,c,i,t,m & (13)\\
I_k &= N_k H_k & (14)
\end{aligned}$$

Here, R_k is the size of the *R*epertoire, or vocabulary, of words and symbols used in $\mathcal{I}_k$, $p_{k,i}$ is the relative frequency in usage of the i-th word or symbol in that repertoire, and N_k is the total number of words and symbols used. Since words and symbols represent first-order chunking by humans, the information first-order entropy measures should correlate strongly with information measures based on chunking. Evaluation of higher-order entropy (phrases, syntactic forms, *etc.*) may be appropriate for study at a later date.

Segments of documents that are included from other sources should not be counted this way, because the apparent information content would be higher than that actually supplied by humans (this time) for its reuse. If such portions can be handled separately, the true human input involvement can more accurately be approximated.

We similarly characterize the inherent input content $\mathcal{I}^*$ and output yield $\mathcal{Y}^*$ in terms of the features of the extensible language. Let R be the number of unique operators and operands that already exist in the current refinery language repertoire, or vocabulary. This number will include both the basic set of built-in functions, as well as every function that has so far been made available to the refinery for reuse (every new function produced is a candidate for reuse, if applicable and feasible). Next, let n denote the number of unique refinery operators of this repertoire actually required for implementing the current application. Then, let d signify the actual number of unique input/output data operands appearing in $\mathcal{I}^*$, and let N be the total number of operators and data operands appearing in $\mathcal{I}^*$. Finally, let $\overline{Y}^*_n$ represent the average inherent yield of the n refinery operators invoked by $\mathcal{I}^*$.

The inherent product information $\mathcal{I}^*$ is just sufficient to specify the product yield; in this, it is a translation of $\mathcal{I}_r$ into specific refinery terms. It specifies the needed functions of the repertoire, the inputs and outputs of each, and the integration of these elements into an appropriate sequence of instructions. We note, then, that $\mathcal{I}^*$ is refinery-dependent, because it depends upon the richness of the repertoire at the time of use. To a first-order approximation, $\mathcal{I}^*$ will be equivalent[4] to N instances of $n+d$ unique operator/operand types arranged in proper order. The minimum average number of bits needed to specify any one of the R operators of the current refinery or d data elements of the current operand vocabulary is the first-order entropy H^* of the refinery and data repertoire. Thus, in analogy with Eq. 14,

$$\begin{aligned} I^* &= NH^* = -N\sum_{i=1}^{R+d} p_i \log_2 p_i & (15)\\ &\leq N\log_2(R+d) & (16)\end{aligned}$$

However, since usage statistics of the refinery and ensemble of applications are unknown at this time, the measure above can only be approximated. For practicality and consistency across languages, the size of the inherent product specification will hereafter in this work be approximated[5] by its upper bound above, also known as the Halstead program volume [5],

$$I^* = N\log_2(R+d) \qquad (17)$$

[4] One may need to normalize I^* across semantically equivalent syntactic constructions of the refinery language. For example, the C language form "x = x + 1" contains 5 tokens, whereas the form "x++" contains only 2. The information content of the two is the same.

[5] Since I^* only appears in the productivity equation in ratio with Y^*, defined in Eq. 18, which is also evaluated in the same way, error due to this approximation will normally be of second order importance.

Note that language processors, for practicality, generally represent tokens using fixed-bit-length internal representations, rather than by variable, frequency-of-use-derived (entropy based) ones. This practice also requires the use of at least $\log_2(R+d)$ bits per token.

Finally, we express the size of the inherent functional yield as

$$Y^* = I^* + n\overline{Y}_n^* \tag{18}$$

The software refinery model thus provides absolute relationships among the current refinery vocabulary size and the average yield of those operator modules in the refinery that were used. Note that I^*, Y^*, n, and $\overline{Y}_n^*$ can all be determined as measurable properties of the software refinery and the current application program. The reuse portion of the product yield, $Y^* - I^*$, should be measured in the refinery language that would be used to reimplement it, regardless of the language used originally to implement it.

5 The Productivity Equation

Let W denote the total *work effort* (measured in work months) required to develop an output information product yield $\mathcal{Y}$ from a given information input set $\mathcal{I}$ supplied by humans. *Productivity* is defined here as the inherent functional yield per unit work, in total bits per work month,

$$P = \frac{Y^*}{W} \tag{19}$$

The use of the inherent functional yield, Y^*, in this definition, rather than the actual apparent yield, Y, which also includes data yield and compiler quirks, is quite arbitrary, but conforms to a practice analogous to counting "executable lines of code," as opposed to "total lines of code." Although Y may perhaps be easier to measure than Y^*, it is, nevertheless, an inadequate indicator of productivity because of its compiler dependence: a better compiler would seem to lower productivity[6].

The average rate at which a given population generates information of a specified type is their *mean work capacity*, C, in bits per work month,

$$C = \frac{I}{W_0} \tag{20}$$

where W_0 is that amount of work required to generate the information $\mathcal{I}$ in an ideal environment where locating existing information, capturing new ideas, and preparing these for use are immediate (*i.e.*, W_0 is measured as the actual work effort minus the location, capture, and preparation effort). C conceptually,

[6] This fact was pointed out to the author by Robert D. Tausworthe of Hewlett-Packard, Inc.

then, is a function of problem complexity, human intellect, experience, skill, motivation, work conditions, staff interaction, and emotional and psychological factors.

We know from experience that human capacity has a limit, so we define the ***potential information capacity***, C_0, as the ideal value of C that could be achieved if the workers were to be relieved of adverse problem, environment, and human factor encumbrances, and were working at a maximum reliable pace. The unitless ratio

$$\mu = \frac{C}{C_0} \leq 1 \tag{21}$$

then represents a ***mental acuity factor***. Since labor wasted in capture and location of information, *etc.*, has been eliminated from μ, it is only independent on environment and tools to the extent that these stimulate individual work capacity. We may note that μ will tend to be greater when $\mathcal{I}$ is produced well within the skill, experience, and understanding of the staff, at a motivated pace of work, and in a smoothly operating and happy organization. However, μ will tend to decrease with other attributes, such as application complexity [1] and staff size [6]. Much of the behavior of μ has been calibrated in various software cost models, where a variation of 500:1 has been noted as necessary to span the range of contributory factors. Consequently, the value of μ for some projects may be on the order of 10^{-3}.

Next, we define ***requirements efficiency***, ρ, as the unitless ratio of inherent product specification and requirements information measures,

$$\rho = \frac{I^*}{I_r} \leq 1 \tag{22}$$

This ratio indicates the level of superfluity between information specifying the as-built product and that contained in requirements information. It is partially a natural characteristic of the requirements and refinery languages being used, but also will depend considerably on the style of the individual(s) writing the requirements, the complexity of the problem, the extent to which fulfilled requirements lead to measurable product specifications, the extent to which stated requirements are fulfilled, the amount and distinguishability of new and reused requirements information, and other factors. Measurements of ρ are needed to calibrate the effects of these factors, and to establish norms for its use as a requirements efficiency indicator. A ball-park figure for ρ based on a few document-to-code size estimates is about 0.1.

The ratio of requirements information to total input information reflects the relative degree to which design, coding, test, and management information are required from humans for a given problem. The ratio of W_0 to W is the effort efficiency in location, capture, and preparation of information. Together, these ratios express the efficiencies of methods, tools, and aids relative to an ideal environment. Labor-saving methods, tools, and aids are those that tend to reduce the amount of *effort* required to generate, capture, or prepare a given amount of information. Examples are word processors, design languages, automated graphics, and data dictionaries. Information-reductive methods, tools, and aids are those that tend to reduce the ***amount of information*** that is required to be generated by humans. Examples here are symbolic notation, automated design assistants, and test case generators.

We combine these two effects into the ***tool factor***, τ, defined as the unitless ratio

$$\tau = \left(\frac{I_r}{I}\right)\left(\frac{W_0}{W}\right) \leq 1 \tag{23}$$

This coefficient reveals the amount of human information, and thus labor, that potentially can be eliminated by methodology, automation, and practice. It provides a simple means by which the effectiveness of solution methods, tools, and engineering processes can be quantified by actual measurements. Note that τ is very likely to be influenced by the ***amount*** of information that must be processed; the greater I is, the greater the difficulty of the human task in coping with it. Thus, we may expect to see the effectiveness of well-designed tools increase as the size and complexity of the project they are applied to increase. A rough estimate of τ from some document page and approximated human effort ratios is about 0.01.

Finally, the refinery ***language advantage***, λ, is defined as the unitless ratio of the reused portion of the output functional yield to the minimum product specification:

$$\lambda = \frac{Y^* - I^*}{I^*} = \frac{n\overline{Y}_n^*}{I^*} \tag{24}$$

This coefficient is quantifiable from token and vocabulary counts in the current refinery model. It represents the information gain factor due to reuse, and signifies how large a product yield can be generated from a minimum product specification in a given refinery environment. Because it is a unitless ratio, λ should be less dependent on a particular refinery than are I^* and Y^* individually, since common tendencies tend to cancel out. A value on the order of about 15 was measured for a group of small C programs using the ANSII standard library functions.

The productivity equation then follows straightforwardly:

$$\begin{aligned} P &= C_0\mu\rho\tau(1+\lambda) && (25) \\ &< C_0(1+\lambda) && (26) \end{aligned}$$

The productivity formula is intuitive: the smallest sufficient requirements definition, the most effortless implementation, and the most propitious usage of tools and methodologies yield the highest advantage; reuse of previous products as new available refinery features yields a higher language advantage.

The upper bound above would be replaced by equality under the condition $\mu\rho\tau = 1$, a situation clearly requiring the existence of automatic programming.

The bound thus shows that the effectiveness of automated programming environments will be determined by the extent of reuse of components in the refinery. Moreover, *the only route to unlimited productivity growth is through the effective reuse of increasingly larger and larger software components.*

6 Language Advantage Trends

It is a remarkable fact that there are statistical laws in natural and computer languages that relate the total number of occurrences of language token types (word types in natural language, and operators and operands in computer languages) to the vocabulary of distinct types used. Laws of this nature were first studied by Zipf [7] in the 1930's in connection with natural languages. Others, notably Halstead [5], Shooman [8], Laemmel [9], Gaffney [10], and Albrecht [11], have extended the study to computer languages and specifications.

The assumption of the method is that the specifications and the programs that embody those specifications are two descriptions of the same thing. Knowledge of one correlates with knowledge about the other. For example, it is reasonable to expect that a statement of basic requirements for a program includes an itemization of its inputs, processing, and outputs, viewed externally. This external statement translates, through the works of Zipf, Halstead, and the other authors cited above, into approximate measures of the output product yield. These measures generally agree within about a factor of 2; hence, we introduce a factor ζ to account for the difference between Zipf's first law and the true refinery model token length characteristic.

Zipf's first law, for example, predicts the approximate token length $\hat{N}$ of $\mathcal{I}^*$ as the value

$$\hat{N} = (n+d)[\gamma + \log(n+d)] \tag{27}$$

where γ is the Euler constant, $\gamma = 0.57721\ldots$. The factor $\zeta = \hat{N}/N$ makes the equation exact, by definition:

$$N = \frac{1}{\zeta}(n+d)[\gamma + \log(n+d)] \tag{28}$$

The token-length correction factor ζ fluctuates from program to program, but ranges approximately between 0.5 and 2.

The refinery language advantage, therefore, is

$$\begin{aligned} \lambda &= \frac{\zeta n \overline{Y}^*_n}{(n+d)\log_2(R+d)[\gamma + \log(n+d)]} && (29) \\ &< \frac{\zeta \overline{Y}^*_n \log 2}{\log n \ \log R} && (30) \\ &< \frac{\zeta \overline{Y}^*_n \log 2}{\log^2 n} && (31) \end{aligned}$$

which, as may be noted, is limited only by average utilized module yield and vocabulary size. As they stand, these expressions are not statistical: λ, ζ, $\overline{Y}^*_n$, and n are determined by the particular program. Averaging λ over an ensemble of programs would yield a statistical bound, however, of the form

$$\overline{\lambda} < \frac{\overline{\zeta}\,\overline{Y}^*_{\overline{n}} \log 2}{\log^2 \overline{n}} \tag{32}$$

for $\overline{\lambda} = E(\lambda)$ and appropriately defined $\overline{\zeta}$ and $\overline{n}$. This statistical form of the bound reveals that, in order for the refinery language advantage (and thus, productivity) to grow without bound, the average yield of refinery modules being used by applications must grow faster than the square of the logarithm of the number of refinery modules being used. That is, it must happen that modules of increasingly higher yields are regularly added to the refinery and regularly used. *A software refinery with a static, non-expanding library imposes a fixed productivity limit on its workers.*

7 Future Work

The work reported here is a part of the newly begun NASA Initiative in Software Engineering (NISE), and is coordinated with other NISE investigations, notably the development of a dual life-cycle paradigm (separating, but interrelating management and engineering processes), the development of a dynamic software life-cycle process simulator, behavioral researches into the performance of humans in the software process, and the synthesis of effective supporting methodologies, tools, and aids.

This first publication reveals only a few rudimentary aspects of the software life cycle process, here modeled as productivity channels refining crude information into highly distilled products. The principal results apply only to the implementation channel, or software refinery. The effects of information noise, the stochastic behavior of people, the detailed character of the other individual component channels, and the dynamic behavior of interacting channels remain to be analyzed and validated.

For the implementation channel, near-term work remains to evaluate C_0, μ, ρ, τ, and λ in a static, low-noise context. Insight into C_0 and μ may be sought in human behavioral research journals. Later work may involve experiments in collaboration with academic researchers.

Typical ρ and τ values may be determined by measurement of documents and programs in existing project libraries for which effort statistics are available; regression with perceived contributory factors would then quantify effects and suggest avenues for productivity improvement. Studies of τ and ρ may be expected to calibrate benefits of selected methodologies and tools.

Still other studies remain to examine the statistical behavior of λ as a function of the refinery size and reuse policy, to determine whether there are natural limits to productivity growth, and thus, to resolve the question posed by the upper bound in Eq. 31 above.

Further research will quantify the behavior of the other component channels of the production life-cycle model, as well as the dynamic interaction of information flows in the model, notably those within the critical loop shown in Figure 2.

8 Conclusion

This publication has developed a model of the software implementation process that formulates productivity as a product of tangible, definite, measurable, and meaningful factors. The model characterizes productivity as stemming from five weakly interrelated factors: human information capacity, mental acuity, requirements specificity, methodology and tool efficiency, and refinery language advantage. Each of these factors was shown to have absolute, explicit, and measurable bounds: human performance is limited by inherent human channel capacity and by the degree of mental acuity that can be achieved toward realizing that capacity. Requirements efficiency is limited by the minimum as-built product specifications and the extent to which requirements specifications can be freed from extraneous, superfluous material. The effectiveness of tools and methodologies is limited to the amount of human (labor) input that can be avoided. And finally, the effectiveness of a programming environment is limited by the average growth in yield of modules in that environment.

These factors serve as absolute standards for comparison purposes: μ reveals how well the staff is meeting its potential; ρ expresses the level of superfluity of requirements; τ quantifies the effectiveness of methodologies, tools, and aids; and λ indicates the power of the refinery. Use of these standards will lead to meaningful tradeoffs and, potentially, to an eventual optimized software life cycle.

References

[1] Boehm, Barry W., "Improving Software Productivity," *Computer*, IEEE Computer Society, Vol. 20, No. 9, 1987, pp. 43-57.

[2] Brooks, Fred P., "No Silver Bullet—Essence and Accidents of Software Engineering," *Proc. IFIP Congress 1986*, North-Holland, 1986, pp. 1069-1076.

[3] Shannon, Claude E., "Communication in the presence of noise," *Proceedings of the I.R.E.*, Vol. 37, 1949, pp. 10-21.

[4] Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capability for Processing Information," *Psychology Review*, March 1956, pp. 81-97.

[5] Halstead, M. H., *Elements of Software Science*, Elsevier North-Holland, Inc., New York, NY, 1977.

[6] Brooks, Fred P., *The Mythical Man Month*, Addison-Wesley Publishing Co., Reading, MA, 1975.

[7] Zipf, G. K., *The Psychobiology of Language: An Introduction to Dynamic Philology*, Houghton Mifflin, Boston, MA, 1935.

[8] Shooman, M. L., *Software Engineering*, McGraw-Hill Book Co., New York, NY, 1983.

[9] Laemmel, A. E., and Shooman, M. L., "Statistical (Natural) Language Theory and Computer Program Complexity," Polytechnic Institute of New York, Report POLY-EE/EP-76-020, August, 1977.

[10] Gaffney, J. E., "Software Metrics: A Key to Improved Development Management," *Computer Science and Statistics: Proc. of the 13*th *Symposium on the Interface*, Springer-Verlag, New York, NY, pp. 211-220.

[11] Albrecht, A. J., and Gaffney, J. E., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. on Software Engineering*, Vol. SE-9, No. 6, November 1983, pp. 639-648.